# A $k$-Opt Based Constraint for the TSP

**Nicolas Isoart**
Université Côte d'Azur, Nice, France

**Jean-Charles Régin**
Université Côte d'Azur, Nice, France

## Abstract

The LKH algorithm based on $k$-opt is an extremely efficient algorithm solving the TSP. Given a non-optimal tour in a graph, the idea of $k$-opt is to iteratively swap $k$ edges of this tour in order to find a shorter tour. However, the optimality of a tour cannot be proved with this method. In that case, exact solving methods such as CP can be used. The CP model is based on a graph variable with mandatory and optional edges. Through branch-and-bound and filtering algorithms, the set of mandatory edges will be modified. In this paper, we introduce a new constraint to the CP model named mandatory Hamiltonian path constraint searching for $k$-opt in the mandatory Hamiltonian paths. Experiments have shown that the mandatory Hamiltonian path constraint allows us to gain on average a factor of 3 on the solving time. In addition, we have been able to solve some instances that remain unsolved with the state of the art CP solver with a 1 week time out.

**2012 ACM Subject Classification** Theory of computation → Constraint and logic programming

**Keywords and phrases** TSP, $k$-opt, 1-tree, Constraint

**Digital Object Identifier** 10.4230/LIPIcs.CP.2021.30

**Funding** This work has been supported by the 3IA Côte d'Azur with the reference number ANR-19-P3IA-0002.

## 1 Introduction

The Traveling Salesman Problem (TSP) is a widely studied graph theory problem with a simple statement: find a minimum cost cycle in a graph visiting all nodes. Unfortunately, solving a TSP is not as easy as stating it: finding the optimal solution of the TSP is NP-Hard.

Heuristics allow one to find non-proved optimal solutions of the TSP in reasonable solving times. The most efficient heuristic solving the TSP is the Lin-Kernighan-Helsgaun (LKH) algorithm [17, 12]. It starts from a tour that is not optimal and iteratively improves the tour with one of the most popular tour improvement algorithms: the local search algorithm $k$-opt [18]. It consists in finding $k$ edges in a given tour such that swapping them create a cheaper tour. Unfortunately, the $k$-opt algorithm has a time complexity in $O(n^k)$ such that $n$ is the number of nodes in a graph. In order to obtain an efficient algorithm, they suggest many improvements such as using a variable $k$ and not considering all the swaps of size $k$ but only the most "promising" swaps.

Exact algorithms allow one to find optimal solutions of the TSP. In practice, they are usually much slower than heuristics because of the optimality proof. The most efficient method solving the "pure" TSP is the specialized solver Concorde [1] based on MIP methods. It is mainly based on the relaxation of the integrity and subtour constraints of the TSP model. In addition, the cutting plane method [5] is used in order to correct structural defects of the intermediate solutions obtained by this relaxation. It proceeds by iteratively generating constraints that are violated by the solution of the relaxed problem. Among them, there are the well-known Comb inequalities. However, no polynomial time algorithm is known at this time to detect whether a solution of the relaxed problem violates a Comb inequality. Therefore, many polynomial time algorithms have been developed in order


to handle particular cases [6, 8, 4, 16]. In addition, Concorde embed other sophisticated techniques such as local cuts. Note that Concorde outperforms the other exact solving methods when considering large graphs. However, CP method is competitive with Concorde for small-medium size graphs [2]. In addition, a TSP is often combined with other constraints. For instance, precedence constraints, TSPTW where there is a time window to visit a node. For these problems, Concorde is not well suited whereas the CP method is a good candidate because it is more robust to side constraints. Nowadays, the most efficient method solving the TSP in CP is the Weighted Circuit Constraint (WCC) [2] in combination with the structural $k$-cutset constraint [13]. The optimization part of the WCC is based on the Lagrangian Relaxation (LR) of Held and Karp [10, 11]. The lower bound of the LR is computed by selecting a node $x$ with its two lowest cost neighbors and a minimum spanning tree in the graph without $x$, *i.e.* it is a 1-tree. If a minimum 1-tree is found such that all its nodes have exactly two neighbors, then an optimal solution is obtained. Thus, the 1-tree is derived through the LR process until an optimal solution is obtained. However, optimally solving the TSP with a LR only can be extremely slow. Thus, the WCC integrates filtering algorithms based on the cost of the edges, the 1-tree cost and a degree constraint on the nodes. In addition, the $k$-cutset constraint is based on the graph structure. It considers the cutsets of the graph containing $k$ mandatory edges and deduces structural filtering. In contrast to heuristic methods, the CP method does not improve a tour but builds an optimal tour. Indeed, the CP model imposes some edges through a branch and bound. Those edges, named mandatory edges, can form paths. Therefore, the purpose of the CP method is to find a tour going through these edges. However, finding an optimal solution can be impossible. For instance, it happens when a path is not itself optimal. Thus, it finds solutions that are suboptimal.

In this paper, we define the mandatory Hamiltonian path constraint that uses the $k$-opt algorithm on the mandatory paths. More precisely, let us define $p$, a path composed of mandatory edges going from $s$ to $t$ through a set of nodes $X'$. If $p$ can be improved by another path $p'$ going from $s$ to $t$ through $X'$, then $p$ cannot belong to an optimal solution. In addition, we define a filtering algorithm removing edges: if a path can be improved when an edge is added to it, then it cannot exist an optimal solution simultaneously containing that path and that edge.

This article is organized as follows: first, we recall some concepts of graph theory. Then, we introduce the TSP in CP with the $k$-cutset constraint and the tour improvement algorithms. Next, we define the mandatory Hamiltonian path constraint and its incremental version. Finally, we discuss some experiments and we conclude.

## 2 Preliminaries

### 2.1 Definitions

The definitions of graph theory are taken from Tarjan's book [21].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** $X$ and an **arc set** $U$, where every arc $(x_i, x_j)$ is an ordered pair of distinct nodes. We note $X(G)$ the set of nodes of $G$ such that $n = |X(G)|$ and $U(G)$ the set of arcs of $G$ such that $m = |U(G)|$. In addition, $U(i)$ is the set of adjacent edges of $i$. The **cost** of an arc is a value associated with the arc. An **undirected graph** is a digraph such that for each arc $(x_i, x_j) \in U$, $(x_i, x_j) = (x_j, x_i)$. If $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ are graphs, both undirected or both directed, $G_1$ is a **subgraph** of $G_2$ if $X_1 \subseteq X_2$ and $U_1 \subseteq U_2$. A **path** from node $x_1$ to node $x_t$ in $G$ is a list of nodes $[x_1, \ldots, x_t]$ such that $(x_i, x_{i+1})$ is an arc for $i \in [1..k-1]$. The path **contains** node

$x_i$ for $i \in [1..k]$ and arc $(x_i, x_{i+1})$ for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $x_1 = x_t$. A cycle is **Hamiltonian** if $[x_1, \ldots, x_{k-1}]$ is a simple path and contains every node of $X$. The **cost** of a path $p$, denoted by $w(p)$, is the sum of the costs of the arcs contained in $p$. For a graph $G$, a solution to the **traveling salesman problem (TSP)** in $G$ is a Hamiltonian cycle $HC \in G$ minimizing $w(HC)$. An undirected graph $G$ is **connected** if there is a path between each pair of nodes, otherwise it is **disconnected**. The maximum connected subgraphs of $G$ are its **connected components**. A **tree** is a connected graph without a cycle. A tree $T = (X', U')$ is a **spanning tree** of $G$ if $X' = X$ and $U' \subseteq U$. The $U'$ edges are the **tree edges** $T$ and the $U - U'$ edges are the **non-tree edges** $T$. A **minimum spanning tree** $T = (X', U')$ is a spanning tree minimizing the cost of the tree edges. A partition $(S, T)$ of the nodes of $G$ such that $S \subseteq X$ and $T = X - S$ is a **cut**. The set of edges $(x_i, x_j) \in U$ having $x_i \in S$ and $x_j \in T$ is the **cutset** of the $(S, T)$ cut. A **$k$-cutset** is a cutset of cardinality $k$.

## 2.2 TSP in CP

The current best CP method solving the TSP is a combination of the Weighted Circuit Constraint (WCC) [2] and the structural constraint $k$-cutset [13]. The WCC is mainly based on the 1-tree Lagrangian Relaxation (LR) of Held and Karp [10, 11]. Intuitively, the LR derives a lower bound of the TSP (here, the 1-tree) until a solution of the TSP is found. A 1-tree is a minimum spanning tree in $G = (X - \{x\}, U)$ such that $x \in X$ is connected by its two nearest neighbors to the minimum spanning tree. Thus, a 1-tree covers the whole graph with $n$ edges and a single cycle. In addition, if the 1-tree satisfies the degree constraint (each node of the 1-tree has exactly two neighbors), then the 1-tree is an optimal solution of the TSP. Therefore, the goal is to minimize the number of nodes that violate the degree constraint in the 1-tree. To do so, this constraint is integrated into the objective function and a Lagrangian multiplier $\pi_i$ is associated to each node $i$. Let $d_i$ be the degree of the node $i$ in the 1-tree. For each node $i$ of the graph, if $d_i < 2$, then $\pi_i$ is decreased. Otherwise, if $d_i > 2$, then $\pi_i$ is increased. Next, the edge cost $w((i, j))$ is modified such that $w'((i, j))$ is the modified cost and $w'((i, j)) = w((i, j)) + \pi_i + \pi_j$. Finally, we obtain an optimal solution of the TSP by computing a succession of 1-trees and modifying the edge costs.

However, experiments shown a very slow convergence toward the optimal solution. Thus, the WCC integrates the following filtering algorithms based on the costs:

- If an edge $e$ does not belong to any 1-tree with cost smaller than a given upper bound, then $e$ can be safely deleted.
- If an edge $e$ belongs to all 1-trees with cost smaller than a given upper bound, then $e$ is mandatory.

Moreover, the WCC integrates a structural constraint imposing that each node has exactly two neighbors (the degree constraint).

Next, for each cutset of size $k$, the $k$-cutset constraint imposes that an even number of edges is mandatory. In practice, the study is limited to $k \leq 3$ since the given algorithm has a complexity growing with $k$. In addition, the interaction of the filtering algorithms and the convergence of the Lagrangian relaxation is not straightforward. Thus, Isoart and Régin [14] introduced an adaptive method in order to improve the overall solving times.

About the search strategy, it consists in making a binary search where a left branch is an edge assignment and a right branch is an edge removal. More precisely, we use the search strategy LCFirst of Fages et al. [7] which is an interpretation of Last Conflict heuristics [9, 15] for graph variables. It selects one edge in the graph according to a heuristic and keeps branching on one extremity of this edge until the extremity is exhausted. Note that it keeps

branching even if a backtrack occurs. Thus, it is a highly dynamic search strategy that learns from previous choices. Moreover, most of the search strategies are much more efficient (up to an order of magnitude) when LCFirst is used. In practice, we observe that using LCFirst strongly interferes with the Lagrangian relaxation and filtering algorithms.

In addition, the WCC uses a single undirected graph variable where all nodes are mandatory. Without loss of generality, we note $O$ the set of optional edges, $M$ the set of mandatory edges and $D$ the set of deleted edges such that $O \cup M \cup D = U$, $O \cap M = \emptyset$, $O \cap D = \emptyset$ and $M \cap D = \emptyset$. Thus, the purpose of the CP is to find a TSP in the input graph $G_{init} = (X, M, O)$ such that $M$ is a growing set and $O$ is a shrinking set. When a solution is found, $|M| = n$ and $O = \emptyset$.

For the sake of clarity, we define $G_{solve} = (X, M', O')$ the current graph such that $M \subseteq M' \subseteq (O \cup M)$ and $O' \subseteq O$. In addition, we define $G_{solve}$ the graph $G_{init}$ modified by the search strategy and the filtering algorithms and $G_{mand} = (X, M', \emptyset)$ the graph of mandatory edges. If not specified, we will use these notations and data structures in the next sections.

## 2.3 Tour improvement algorithms

In order to find a TSP, a tour improvement algorithm takes as input a tour and iteratively tries to improve it. The most popular tour improvement algorithms are the local search algorithms 2-opt and 3-opt.

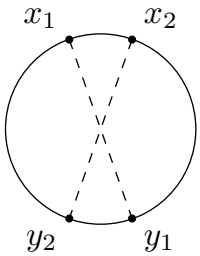


**Figure 1** An example of 2-opt. The circle represents a tour and the dashed lines are the suggested move for the pair of edges $((x_1, x_2), (y_1, y_2))$.

The idea of 2-opt is pretty simple. Given a tour $T$, for each pair of edges $(e_1, e_2)$ in $T$, if replacing $(e_1, e_2)$ by another pair of edges $(e_3, e_4)$ of $T$ leads to a connected and shorter tour, then we can replace $(e_1, e_2)$ with $(e_3, e_4)$ in $T$. We name such a replacing procedure a move. Note that some heuristics are looking for the best improving move before applying the replacement of a move. In addition, for each pair of edges, there is only one move reconnecting the graph that is not the null move. The iteration on pairs leads to a time complexity in $O(n^2)$. Figure 1 shows an example where $e_1 = (x_1, x_2)$, $e_2 = (y_1, y_2)$ and the move is $e_3 = (x_1, y_1)$, $e_4 = (x_2, y_2)$.

For the 3-opt algorithm, instead of choosing pair of edges, we choose a triplet of edges and, as for 2-opt, we search for moves reducing the overall cost of the tour. In that case, there are seven ways to reconnect the graph. Note that three of them are simple 2-opt (that is a combination with one edge of the triplet not moved). Thus, 3-opt allows checking more sophisticated combination than 2-opt and then can potentially find better moves. However, it leads to an algorithm with a time complexity in $O(n^3)$. Figure 2 shows an example of all 3-opt moves that are not 2-opt.

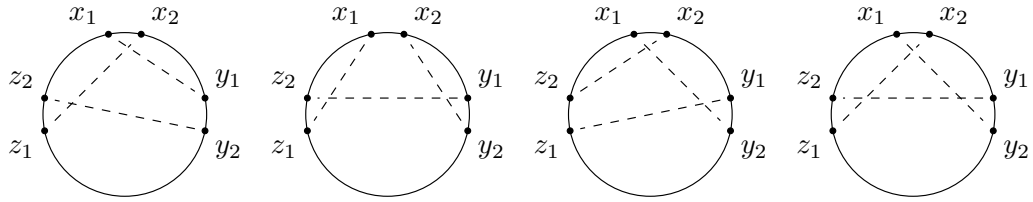


**Figure 2** An example of 3-opt. The circle represents a tour and the dashed lines are the suggested move for the triplet of edges $((x_1, x_2), (y_1, y_2), (z_1, z_2))$.

Naturally, the 2-opt and the 3-opt algorithms can be generalized to the $k$-opt algorithm with a time complexity in $O(n^k)$. Experiments have shown that increasing the value of $k$ improves the quality of the tours but slows down solving times. Thus, some methods [19, 3] consider some 3-opt and/or 4-opt, but not all, in order to reduce the time complexity and speed up the solving times.

Moreover, Lin and Kernighan suggested to use a variable $k$ while solving [17] in order to include larger moves. The algorithm is therefore more complex but it greatly improves the results (tour quality and solving times). To do so, they suggested several rules. First, they are looking for the most promising permutations only. Next, they allow improving $k$-opt moves that can be built from a sequence of 2-opt moves such that some moves do not improve the tour. These moves are much more complex and provide better moves than a simple run of the 2-opt algorithm. In order to make this algorithm extremely efficient, Helsgaun [12] has remarkably refined most of the rules given by Lin and Kernighan [17]. Nowadays, the Lin-Kernighan-Helsgaun algorithm is considered as one of the most efficient heuristic solving the TSP and therefore it is embedded in most of the exact methods.

In this paper, we integrate 2-opt and 3-opt concepts into CP. Unlike tour improvement algorithms, the CP model does not have a tour to improve. However, the CP model has mandatory edges that can form paths and try to find a tour going through these paths. We then search for 2-opt and 3-opt in the paths of mandatory edges.

## 3 Mandatory Hamiltonian path constraint

We note $M'(i)$ (resp. $O'(i)$) the set of mandatory (resp. optional) edges having $i$ for extremity in $M'$ (resp. $O'$). For each node $i$, $|M'(i)| \leq 2$ because of the degree constraint. Thus, the mandatory edges form disjoint paths. Without loss of generality, we assume that the current assignment of the $G_{solve}$ is consistent with the degree constraint.

▶ **Definition 1.** *A mandatory Hamiltonian path $p$ is a path such that $p$ is a Hamiltonian path in a subgraph of $G_{solve}$ and for each edge $e = (x_i, x_{i+1})$ of $p$, $e \in M'$.*

We note $p_1 = [x_1, x_2, \ldots, x_t]$ a mandatory Hamiltonian path of $G_{solve}$.

### 3.1 Consistency Check

In this section, we will study the existence of optimal solutions in $G_{solve}$.

▶ **Definition 2.** *An **alternative path** $p_2 = [x'_1, x'_2, \ldots, x'_t]$ of $p_1$ is a permutation of the nodes of $p_1$ such that $p_1 \neq p_2$, $x_1 = x'_1$, $x_t = x'_t$ and for each $i \in [1, k-1]$, $(x'_i, x'_{i+1}) \in U$.*

Figure 3 shows an example of an alternative path. Thus, an alternative path can be composed of edges in $M \cup O \cup D$, *i.e.* in $U$.

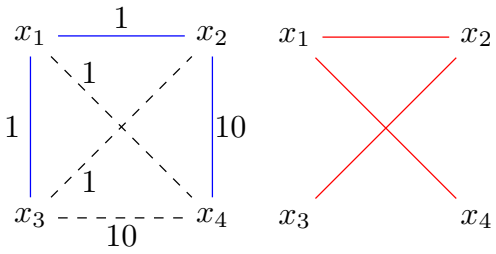


**Figure 3** The left graph is a subgraph of $G_{init}$. The blue edges are from $M$, they form a mandatory Hamiltonian path going from $x_3$ to $x_4$. The dashed edges are from $D$ (the deleted edges). The right graph is an alternative path of the left graph.

▶ **Definition 3.** *The mandatory Hamiltonian path $p_1$ is minimal if and only if there is no alternative path $p_2$ of $p_1$ such that $w(p_2) < w(p_1)$.*

In Figure 3, the mandatory Hamiltonian path is not minimal. The right graph represents an alternative path with a cost of 4 whereas the mandatory Hamiltonian path has a cost of 12. The idea is to search for a non-minimal mandatory Hamiltonian path $p_2$ in the connected components of $G_{mand} = (X, M', \emptyset)$. In Proposition 4, we show that if such a path $p_2$ exists, then the cost of the TSP in $G_{solve} = (X, M', O')$ is greater than the cost of the TSP in $G_{init} = (X, M, O)$.

▶ **Proposition 4.** *If there is a mandatory Hamiltonian path $p_1$ that is not minimal, then $p_1$ cannot belong to any solution of $TSP(G_{init})$.*

**Proof.** Given $w(TSP(G_{init} + p_1))$ the cost of $TSP(G_{init})$ such that $p_1$ is in the solution. If there is no solution for $TSP(G_{init} + p_1)$, then $p_1$ cannot belong to any solution $TSP(G_{init})$. Otherwise, if $p_1$ is not minimal, then there is an alternative path $p_2$ of $p_1$ such that $w(p_2) < w(p_1)$. Thus, $w(TSP(G_{init} + p_2)) < w(TSP(G_{init} + p_1))$ and therefore $p_1$ cannot belong to any $TSP(G_{init})$. ◀

In the context of a CP solver, if there is a mandatory Hamiltonian path $p$ that is not minimal, then from Proposition 4 we can trigger a failure because the current solution is not minimal. Moreover, it raises a question: how do we verify if a mandatory Hamiltonian path is minimal? A first algorithm consists in checking all the possible permutations for each mandatory Hamiltonian path. Unfortunately, checking all the permutations leads to an impractical algorithm. However, a large number of heuristics improving tours have been designed. Among them, there are the ones introduced in Subsection 2.3. Thus, we can use any of these heuristics on the mandatory Hamiltonian paths. If it finds an improvement, then a mandatory Hamiltonian path is not minimal and therefore we can trigger a failure.

In this paper, we use $k$-opt heuristics as tour improvement since they are very efficient and easy to implement. In Section 4, we will show that 2-opt and 3-opt are enough in order to obtain good results.

▶ **Definition 5.** *Given the set of mandatory Hamiltonian paths $P$ and an integer $k$. For each mandatory Hamiltonian path $p \in P$, the mandatory Hamiltonian path constraint ensure that there is no alternative path $p'$ obtained by swapping $k$ edges of $p$ such that $w(p') < w(p)$.*

Therefore, we define the mandatory Hamiltonian path constraint in Definition 5 such that if its consistency is not verified, then we trigger a failure.

**Algorithm 1** Consistency check of the mandatory Hamiltonian paths.

```
1 ConsistencyCheck (G_init, G_mand, k)
    Input: The initial graph G_init, the graph of mandatory edges G_mand and an
           integer k.
    Output: A Boolean specifying whether G_mand contains a mandatory
            Hamiltonian path that is not minimal.
2   P ← computeMandatoryHamiltonianPaths(G_mand) ;
3   foreach path p ∈ P do
4     if k-optPath(G_init, p) then
5       return False ;
6   return True ;
```

In Algorithm 1, we introduce an implementation of the algorithm checking the consistency of the mandatory Hamiltonian path constraint. We assume that $k$-optPath$(G_{init}, p)$ returns true if and only if the mandatory Hamiltonian path constraint with the given $k$ is consistent. Internally, $k$-optPath$(G_{init}, p)$ uses a $k$-opt heuristic. Then, for each mandatory Hamiltonian path $p$, we run $k$-optPath$(G_{init}, p)$ in $O(|p|^k)$.

▶ **Proposition 6.** *Given $P$ the set of mandatory Hamiltonian paths. Then, $\sum_{p \in P} |p| \leq n$ and $|P| \leq n$.*

**Proof.** By definition, each node can only be contained in one mandatory Hamiltonian path and there are $n$ nodes in $G_{solve}$. Thus, $\sum_{p \in P} |p| \leq n$. In addition, if each node is contained in a different path, then there are $n$ paths and then therefore $|P| \leq n$. ◀

Each $p$ of $P$ are disjoint. From Proposition 6, the sum of $|p|$ for all $p \in P$ is lower or equal to $n$. Finally, the time complexity of Algorithm 1 is in $O(\sum_{p \in P} |p|^k) < O(n^k)$.

## 3.2 Filtering algorithm

In this section, we will consider that the consistency has been checked. An edge $e = (x_t, x_i)$ is a successor of $p_1$ and an edge $e = (x_i, x_1)$ is a predecessor of $p_1$. In addition, we note $x_i + p_1 = [x_i, x_1, x_2, \ldots, x_t]$ and $p_1 + x_i = [x_1, x_2, \ldots, x_t, x_i]$.

From Proposition 4, we have the two following corollaries:

▶ **Corollary 7.** *For each edge $e \in O'$ such that $e$ is a predecessor of $p_1$, if $i + p_1$ is not a minimal mandatory Hamiltonian path, then $e$ cannot belong to any solution of $TSP(G_{solve})$.*

▶ **Corollary 8.** *For each edge $e \in O'$ such that $e$ is a successor of $p_1$, if $p_1 + i$ is not a minimal mandatory Hamiltonian path, then $e$ cannot belong to any solution of $TSP(G_{solve})$.*

Thus, in order to define a filtering algorithm we are interested in the minimality of $p_1 + i$ and $i + p_1$. If the minimality of $p_1$ has already been checked, then we can avoid all permutations only containing the elements of $p_1$. We impose $i$ to be in the considered permutations and we look for permutations of size $(k-1)$ in $p_1$. Then, for a mandatory Hamiltonian path $p$ and a single successor or predecessor, we can filter the edge in $O(|p|^{k-1}) < O(n^{k-1})$. Performing the filtering for all predecessors and successors of $p$ can be done in $O(|O'(p)||p|^{k-1}) < O(n^k)$. From Proposition 6, there can be at most $n$ paths and the sum of the size of all paths is smaller than or equal to $n$. Thus, from Corollary 7 and 8 a we can filter the edges of all paths with a complexity in $O(n^{k+1})$. Note that the number of checked permutations in practice is much smaller.

A mandatory Hamiltonian path $p_1$ can have a successor or a predecessor $e$ connecting another mandatory Hamiltonian path $p_2$. Then, adding $e$ to the solution leads to a minimality check in $p_1 + p_2$. We can then extend the two previous corollaries:

▶ **Corollary 9.** *For each edge $e = (x_i, x_1) \in O'(x_1)$, if it exists $p_2 = [x'_1, x'_2, \ldots, x'_t]$ a mandatory Hamiltonian path of $G_{solve}$ such that $x_i = x'_t$ and $p_2 + p_1$ is not a minimal mandatory Hamiltonian path, then $e$ cannot belongs to any $TSP(G_{solve})$.*

▶ **Corollary 10.** *For each edge $e = (x_i, x_t) \in O'(x_t)$, if it exists $p_2 = [x'_1, x'_2, \ldots, x'_t]$ a mandatory Hamiltonian path of $G_{solve}$ such that $x_i = x'_1$ and $p_1 + p_2$ is not a minimal mandatory Hamiltonian path, then $e$ cannot belongs to any $TSP(G_{solve})$.*

Given a mandatory Hamiltonian path $p_2$ of $G_{solve}$ connected to $p_1$ with $e \in O'$. Then, we have to check if $p_1 + p_2$ is minimal in order to determine whether $e$ can be in a solution of $TSP(G_{solve})$. It can be done with Corollary 9 and 10 in $O((|p_1|+|p_2|)^k - |p_1|^k - |p_2|^k) < O(n^k)$. The number of predecessors and successors of $p_1$ is at most $2n$. If $P(p_1)$ is the set of mandatory Hamiltonian paths such that each path of $P(p_1)$ is connected to $p_1$ with a successor or a predecessor of $p_1$, then the filtering on $p_1$ can be done in $O(\sum_{p_2 \in P(p_1)}(|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+1})$. Given $P$ the set of mandatory Hamiltonian paths. The filtering for all paths of $P$ can be done in $O(\sum_{p_1 \in P} \sum_{p_2 \in P(p_1)}(|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+2})$. Algorithm 2 is a possible implementation.

For the sake of clarity, we will use the following notations in the algorithms:

- $P$: contains all the mandatory Hamiltonian paths of the graph $G_{solve}$.
- $P[i]$: if there is a path $p$ containing the node $i$, then it returns $p$. Otherwise, it returns $i$.
- $p$.first(): returns the first node of the path $p$.
- $p$.last(): returns the last node of the path $p$.

**Algorithm 2** Filtering algorithm for the mandatory Hamiltonian paths.

```
1  Filter (G_init, G_solve = (X, M', O'), P, k)
       Input: The initial graph G_init, a graph G_solve, the set of mandatory
              Hamiltonian paths P and an integer k.
2      foreach p_1 = [x_1, x_2, ..., x_t] ∈ P do
3          foreach edge e = (x_1, j) ∈ O'(x_1) do
4              p_2 ← P(j) ;
5              if p_2.last() ≠ j then reverse(p_2);
6              if (p_1 = p_2 and |M'| ≠ n − 1) or k-optPath(G_init, p_2, p_1) then
7                  O' ← O' − e ;
8          foreach edge e = (x_t, j) ∈ O'(x_t) do
9              p_2 ← P(j) ;
10             if p_2.first() ≠ j then reverse(p_2);
11             if (p_1 = p_2 and |M'| ≠ n − 1) or k-optPath(G_init, p_1, p_2) then
12                 O' ← O' − e ;
```

Given $P$ the set of the mandatory Hamiltonian paths. For each path $p_1 \in P$, we perform the filtering on all the predecessor and successor $e$ of $p_1$. We note $p_2$ the path connected to $p_1$ by $e$ ($p_2$ can be a single node). In addition, we note $k$-optPath$(graph, p_1, p_2)$ the $k$-optPath algorithm considering the permutations of $p_1 + p_2$ such that each permutation contains at

least one element of $p_1$ and at least one element of $p_2$. When two paths are merged, they must be in the right order. If $p_2$ must be inserted in front of $p_1$, then the node $j$ must be the last node of $p_2$. Otherwise, $j$ must be the first node of $p_2$. Thus, $p_2$ is reversed if needed. Note that we can save the reversed path in order to avoid redundant computations. If an improvement is found when $p_1$ and $p_2$ are merged, then from Corollary 9 or 10 the edge $e$ cannot belong to a solution of $TSP(G_{solve})$ and therefore $e$ is removed from the optional edges of $G_{solve}$. In addition, if $p_1 = p_2$ and $|M'| \neq n - 1$, then it exists an edge $e = (i, j)$ such that $i$ and $j$ belong to the same mandatory Hamiltonian path and therefore the edge close a cycle with a size lower than $n$. Thus, adding $e$ to the solution creates a sub-cycle and then $e$ is removed from the optional edges of $G_{solve}$.

## 3.3 Maintenance during the search

In this section, we will consider the incremental aspect of this constraint, *i.e.* the consistency of this constraint or its filtering when some edges become mandatory or deleted. Moreover, we will consider the restoration of the data structures introduced for the incremental aspect when a backtrack occurs. In this study, an edge can be deleted or an edge becomes mandatory.

▶ **Proposition 11.** *Given $G' = (X, M', O'')$ such that $O'' \subseteq O'$. If $p_1$ is minimal, then $p$ is minimal in $G'$.*

**Proof.** The graph $G'$ is the graph $G_{solve}$ such that some edges are deleted. By definition, the deleted edges are in $D$ and the alternative paths can contain edges of $D$. Thus, if $p_1$ is minimal, then $p$ is minimal in $G'$. ◀

From Proposition 11, if we know that all the mandatory Hamiltonian paths of $G_{solve}$ are minimal and then some edges are removed, then the mandatory Hamiltonian paths of $G_{solve}$ remain minimal. In addition, removing some edges does not change the result of the filtering algorithm since new alternative paths cannot be created from removal. Thus, the consistency test and the filtering algorithm are only triggered when there are new mandatory edges.

In the following algorithms, we use the following data structures:

- candidates: a stack of graph nodes such that the nodes are adjacent to edges that can be filtered.
- deltaMand: a set of the new mandatory edges since the last call of the constraint for the current search node.

### 3.3.1 Consistency check

When an edge $e$ becomes mandatory, there are three cases:

- $e$ is not connected to any path and therefore $e$ creates a new path only containing its two endpoints. Note that a mandatory Hamiltonian path with two nodes is necessarily minimal.
- $e$ is connected to a mandatory Hamiltonian path $p$ and therefore $p$ and $e$ are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.
- $e$ is connected to two mandatory Hamiltonian paths $p_1$ and $p_2$ and therefore $p_1$ and $p_2$ are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.

Thus, for consistency check, we only consider the paths that must be merged. In addition, given a new mandatory edge $e$ connecting $p_1$ and $p_2$, we note $p_3$ the merged path of $p_1$ and $p_2$. When two paths are merged, we assume that the minimality check has been performed

on the two paths. Therefore, when the $k$-optPath algorithm is checking the minimality for the path $p_3$, it can avoid the permutations containing either only elements of $p_1$ or only elements of $p_2$. Then, we consider the permutations that contain at least one element of $p_1$ and at least one element of $p_2$.

In Algorithm 3, we give a possible implementation of the incremental algorithm checking the minimality of the mandatory Hamiltonian paths. For each edge $(i, j)$ newly mandatory, we have $p_1$ and $p_2$ the mandatory Hamiltonian paths such that $i$ and $j$ are respectively an extremity of $p_1$ and $p_2$. Therefore, the edge $(i, j)$ merge $p_1$ and $p_2$ and $p_1$ and/or $p_2$ are accordingly reversed. Note that the *candidates* stack is filled for the filtering algorithm. Then, we run the $k$-optPath algorithm in order to find alternative paths in $p_1 + p_2$. Note that we only consider permutations such that each permutation contains at least one element of $p_1$ and at least one element of $p_2$. Finally, if no alternative path is found, $p_1 + p_2$ is a minimal mandatory Hamiltonian path and we merge $p_1$ and $p_2$ in $P$. Otherwise, we return False and a failure is triggered.

**Algorithm 3** Incremental minimality check of the mandatory Hamiltonian paths.

```
1  IncrementalConsistencyCheck (G_init, P, deltaMand, candidates, k)
       Input: The initial graph G_init, the set of mandatory Hamiltonian paths P, the
              set of new mandatory edges deltaMand, candidates a filtering used stack
              and an integer k.
       Output: A Boolean specifying whether P contains a mandatory Hamiltonian
               path that is not minimal.
2      foreach (i, j) ∈ deltaMand do
3          p_1 ← P[i] ;
4          p_2 ← P[j] ;
5          if p_1.last() ≠ i then reverse(p_1);
6          if p_2.first() ≠ j then reverse(p_2);
7          candidates.push(p_1.first()) ;
8          candidates.push(p_2.last()) ;
9          if k-optPath(G_init, p_1, p_2) then
10             return False ;
           // merge p_1 and p_2 in P
11         merge(P, p_1, p_2) ;
12     return True ;
```

The overall time complexity of Algorithm 3 is $O(\sum_{(i,j)\in deltaMand}(|P[i]| + |P[j]|)^k - |P[i]|^k - |P[j]|^k) < O(n^k)$. Note that Algorithm 1 has a time complexity in $O(\sum_{p\in P}|p|^k) < O(n^k)$ when all paths are already merged which is equivalent to $O(\sum_{(i,j)\in deltaMand}(|P[i]| + |P[j]|)^k)$ if paths are not merged. Thus, the incremental algorithm improves the time complexity for checking the minimality of the mandatory Hamiltonian paths.

### 3.3.2 Filtering algorithm

When an edge $e$ becomes mandatory, we have the same three cases as for the consistency check. Thus, we will only consider the merged mandatory Hamiltonian paths in the previous consistency check. More precisely, we will only consider the neighborhood of the first node and the last node of these paths.

Algorithm 4 is a possible implementation of an incremental algorithm performing the filtering. First, we iterate on *candidates*. Every time two paths are merged in Algorithm 3, the first and last nodes of the merged path are pushed in *candidates*. Thus, *candidates* contains the first node and last nodes of all merged paths. In addition, *candidates* may contain some nodes that are "intermediate" merged paths. For example, merging $p_1$ and $p_2$ results in $p_3$ such that $p_3.first() = x$ and $p_3.last() = y$. Then, $x$ and $y$ are pushed in *candidates*. Merging $p_3$ with $p_4$ results in $p_5$ such that $p_5.first() = x'$ and $p_5.last() = y'$. Then, $x'$ and $y'$ are pushed in *candidates*. However, $x$ and $y$ still are in *candidates* while $x$ or $y$ is no longer the first node or the last node of a merged path. Then, while iterations on candidates, we need to avoid these nodes. Finally, if a node $i$ is an endpoint of a mandatory Hamiltonian path, then we check in the neighborhood of the node $i$ (same as for Algorithm 2).

**Algorithm 4** Incremental filtering of the mandatory Hamiltonian paths.

```
1  IncrFiltering (G_init, G_solve = (X, M', O'), P, candidates, k)
       Input: The initial graph G_init, a graph G_solve, the set of mandatory
              Hamiltonian paths P, the stack of nodes to consider for the filtering
              candidates and an integer k.
2      while candidates.isNotEmpty() do
3          i ← candidates.pop() ;
4          p1 ← P[i] ;
5          if i = p1.first() or i = p1.last() then
6              foreach edge e = (i, j) ∈ O'(i) do
7                  p2 ← P[j] ;
8                  if p1.last() ≠ i then reverse(p1);
9                  if p2.first() ≠ j then reverse(p2);
10                 if (p1 = p2 and |M'| ≠ n − 1) or k-optPath(G_init, p1, p2) then
11                     O' ← O' − e ;
```

If $P(i)$ is the set of mandatory Hamiltonian paths such that each path of $P(i)$ is connected with a successor or a predecessor of $P[i]$, then the time complexity of Algorithm 4 is in $O(\sum_{i \in candidates} \sum_{p_2 \in P(i)} (|P[i]| + |p_2|)^k - |P[i]|^k - |p_2|^k)) < O(n^{k+2})$.

### 3.3.3 Restoration

In order to save more computations, we maintain the set $P$ of mandatory Hamiltonian paths. When a backtrack occurs, the difference between the backtracked state and the current state is that some mandatory edges could have been found and therefore some mandatory Hamiltonian paths of $P$ could have been merged. Thus, in order to restore $P$, the merged mandatory Hamiltonian paths should be split. To do so, we define a stack $S$ such that $S$ contains the added mandatory edges from the root to the current state. In addition, we save the size of the stack for each open search node. Then, when a backtrack occurs, we iteratively pop the mandatory $e$ edges of $S$ until the wanted size is obtained. For each $e$, we split the mandatory Hamiltonian path in $P$ containing $e$.

## 4 Experiments

The algorithms have been implemented in Java 11 in a locally developed constraint programming solver. The experiments were performed on Clear Linux with an Intel Xeon E5-2696v2 and 64 GB of RAM. The instances are from the TSPLib [20], a library of reference graphs for the TSP. We rerun experiments ran in Isoart and Régin [13] and exclude instances solved in less than two seconds by the state of the art. In addition, we tried some harder instances from the TSPLib and selected those that did not have reached a time out *t.o.* by both the state of the art and our method. Note that we set *t.o.* to 1 week, that is 604,800 seconds. The name of each instance is suffixed by its number of nodes. In our implementation, the TSP is modeled by the WCC using the CP-based LR configuration introduced in Isoart and Régin [14]. We note "state of the art" the TSP model introduced in Subsection 2.2, "MHP 2-opt" the state of the art combined with the mandatory Hamiltonian path constraint searching for 2-opt and "MHP 3-opt" the state of the art combined with the mandatory Hamiltonian path constraint searching for 3-opt. The search strategy used is LCFirst with the heuristic minDeltaDeg [7] which is also the state of the art. Given $e = (i, j)$ an edge, minDeltaDeg selects the edge with the minimum difference between the sum of the number of optional neighbors of $i$ and $j$ and the sum of the number of mandatory neighbors of $i$ and $j$. Thus, we compare our constraint and the state of the art through the number of backtracks (#bk) and the solving times in seconds in arrays. All considered instances are symmetric graphs. If not specified, we use the implementation given in Algorithm 3 and 4.

Table 1 shows the solving times and the number of backtracks for the state of the art solving method and with 2-opt and 3-opt added to it. In addition, we display a ratio column in order to show the gain factor for each instance by using 2-opt and 3-opt.

For the state of the art, we notice that 4 instances over 32 have reached the time out. For the mandatory Hamiltonian path constraint combined with 2-opt, we notice that only 2 of the 4 instances have reached the time out. Indeed, pr299 is solved in 9,640s and rd400 is solved in 28,122s with 2-opt whereas they remain unsolved in 604,800s with the state of the art.

Most of the time, we notice that the use of 2-opt allows us to improve the solving times. For example, ali535 is improved by a factor of 3.5 in solving time and by a factor of 3.7 in backtracks. Some problems have higher gain factors: d493 gains a factor 6 in solving time and a factor 4.9 in backtracks. Moreover, only pr124 has a degraded solving time when using 2-opt: 2.8s vs 3.3s. Note that there is gain in backtracks 1856 vs 1700.

The mandatory Hamiltonian path constraint combined with 3-opt allow us to obtain an additional improvement to the use of 2-opt only. Indeed, 3-opt can be slower than 2-opt in terms of backtracks/second but it greatly reduce the number of backtracks. Note that this configuration solve all the considered instances. Indeed, pr299 is solved in 3,039s, pr493 is solved in 170,127s, rd400 is solved in 12,352s and u574 is solved in 198,693s with the mandatory Hamiltonian path constraint combined with 3-opt whereas they remain unsolved in 604,800s with the state of the art. We thus obtain great improvement factors on the solving times: $> 199$ for pr299, $> 3.6$ for pr493, $> 49$ for rd400 and $> 3$ for u574. In addition, some instances are solved much faster with 3-opt than with 2-opt: gr666 is solved in 303.293s with the state of the art, 64,391s with 2-opt and 21,853s with 3-opt. Some other instances are solved with almost the same number of backtracks for 2-opt and 3-opt: for ali535 with 2-opt there is 3,148,626bk and there is 3,178,482bk with 3-opt. However, it has a slower solving time with 2-opt than with 3-opt: 24,367s vs 35,026s. This can be due to several reasons: the extra cost of using an algorithm in $O(n^3)$ compared to an algorithm in $O(n^2)$.

**Table 1** General results comparing the mandatory Hamiltonian path constraint combined with 2-opt or 3-opt and the state of the art.

| | State of the art (1) | | MHP 2-opt (2) | | ratio (1)/(2) | | MHP 3-opt (3) | | ratio (1)/(3) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | time(s) | #bk | time(s) | #bk | time | #bk | time(s) | #bk | time(s) | #bk |
| a280 | 7.0 | 2,372 | 6.5 | 2,182 | 1.1 | 1.1 | 10.8 | 3,134 | 0.6 | 0.8 |
| ali535 | 84,929.1 | 11,747,704 | 24,367.5 | 3,148,626 | 3.5 | 3.7 | 35,026.0 | 3,178,482 | 2.4 | 3.7 |
| ch150 | 2.9 | 1,526 | 2.1 | 644 | 1.4 | 2.4 | 1.7 | 392 | 1.7 | 3.9 |
| d198 | 14.0 | 7,192 | 12.6 | 6,062 | 1.1 | 1.2 | 10.4 | 3,694 | 1.4 | 1.9 |
| d493 | 95,916.6 | 13,478,616 | 15,931.0 | 2,778,780 | 6.0 | 4.9 | 31,162.1 | 1,877,298 | 3.1 | 7.2 |
| gil262 | 5,230.2 | 2,254,728 | 3,804.8 | 1,710,410 | 1.4 | 1.3 | 3,833.3 | 1,501,756 | 1.4 | 1.5 |
| gr137 | 3.4 | 1,910 | 1.7 | 706 | 2.0 | 2.7 | 1.5 | 518 | 2.2 | 3.7 |
| gr202 | 2.4 | 886 | 2.0 | 600 | 1.2 | 1.5 | 2.3 | 448 | 1.0 | 2.0 |
| gr229 | 227.0 | 166,378 | 64.6 | 44,696 | 3.5 | 3.7 | 59.1 | 33,336 | 3.8 | 5.0 |
| gr431 | 1,724.8 | 265,698 | 494.8 | 68,432 | 3.5 | 3.9 | 556.3 | 65,100 | 3.1 | 4.1 |
| gr666 | 303,293.4 | 28,432,754 | 64,390.7 | 5,168,402 | 4.7 | 5.5 | 24,853.1 | 1,721,794 | 12.2 | 16.5 |
| kroA100 | 2.0 | 1,270 | 1.2 | 438 | 1.7 | 2.9 | 1.3 | 458 | 1.6 | 2.8 |
| kroA150 | 6.1 | 4,164 | 5.2 | 3,374 | 1.2 | 1.2 | 3.9 | 1,814 | 1.6 | 2.3 |
| kroA200 | 401.0 | 237,806 | 63.8 | 33,166 | 6.3 | 7.2 | 68.2 | 34,058 | 5.9 | 7.0 |
| kroB100 | 5.4 | 4,816 | 2.8 | 2,164 | 1.9 | 2.2 | 1.7 | 972 | 3.2 | 5.0 |
| kroB150 | 262.6 | 247,574 | 30.5 | 23,296 | 8.6 | 10.6 | 21.7 | 16,012 | 12.1 | 15.5 |
| kroB200 | 127.8 | 87,296 | 34.5 | 21,060 | 3.7 | 4.1 | 15.8 | 8,140 | 8.1 | 10.7 |
| kroC100 | 2.0 | 1,470 | 1.1 | 346 | 1.8 | 4.2 | 1.1 | 334 | 1.8 | 4.4 |
| kroE100 | 2.3 | 1,804 | 1.6 | 782 | 1.4 | 2.3 | 1.6 | 824 | 1.4 | 2.2 |
| lin318 | 32.9 | 7,834 | 8.7 | 1,944 | 3.8 | 4.0 | 10.1 | 2,018 | 3.3 | 3.9 |
| pr124 | 2.8 | 1,856 | 3.3 | 1,700 | 0.8 | 1.1 | 2.3 | 1,142 | 1.2 | 1.6 |
| pr136 | 20.4 | 18,684 | 16.2 | 13,886 | 1.3 | 1.3 | 13.1 | 8,598 | 1.6 | 2.2 |
| pr144 | 2.3 | 1,036 | 1.8 | 628 | 1.3 | 1.6 | 1.9 | 594 | 1.2 | 1.7 |
| pr264 | 4.7 | 690 | 4.9 | 508 | 1.0 | 1.4 | 5.1 | 524 | 0.9 | 1.3 |
| pr299 | *t.o.* | *t.o.* | 9,640.5 | 2,710,230 | $> 62.7$ | - | 3,038.7 | 805,344 | $> 199.0$ | - |
| pr439 | *t.o.* | *t.o.* | *t.o.* | *t.o.* | - | - | 170,127.1 | 35,750,706 | $> 3.6$ | - |
| rat195 | 38.5 | 24,274 | 17.9 | 10,286 | 2.2 | 2.4 | 17.8 | 8,560 | 2.2 | 2.8 |
| rd400 | *t.o.* | *t.o.* | 28,121.1 | 6,524,576 | $> 21.5$ | - | 12,351.9 | 2,507,272 | $> 49.0$ | - |
| si175 | 288.5 | 301,102 | 204.5 | 197,968 | 1.4 | 1.5 | 342.6 | 275,870 | 0.8 | 1.1 |
| tsp225 | 121.5 | 65,002 | 116.8 | 59,688 | 1.0 | 1.1 | 51.4 | 24,042 | 2.4 | 2.7 |
| u574 | *t.o.* | *t.o.* | *t.o.* | *t.o.* | - | - | 198,962.7 | 28,269,058 | $> 3.0$ | - |

The Lagrangian relaxation can also be impacted by the filtered edges. However, since 3-opt solves more problems than 2-opt and that on average (if we do not consider the instances the instances that have reached the time out) we obtain a gain of a factor of 2.5 for 2-opt and 3 for 3-opt over the state of the art. Thus, we will consider the version with 3-opt. Note that we also could use some other heuristics such as 2.5-opt that compute 2-opt and some 3-opt. The improvement over the number of backtracks is not as much important as for the 3-opt method but the number of backtracks per second is higher. In practice, we have observed on average a 10% difference on the solving times between 3-opt and 2.5-opt.

In Table 2, we show the impact of the use of the incremental version of the mandatory Hamiltonian path constraint on some instances of Table 1. On this instance set, the incremental version is on average 33% faster than the non-incremental one. With the incremental version, the solving times of somes instances such as d198 are improved of 8% whereas for other instances such as gr229 the solving times are improved of 50%. Thus, the benefit of avoiding recalculations may be interesting for this constraint due to the time complexity of the $k$-opt algorithm.

In Table 3, we are interested in the use of $k$-opt algorithms with $k$ greater than 3. For the number of backtracks, we notice that on average 4-opt is more efficient than 3-opt which is more efficient than 2-opt. In addition, 4-opt and 5-opt achieve similar results. However,

**Table 2** Comparison of solving times for the non-incremental and the incremental version of the mandatory Hamiltonian path constraint.

| | (1) 3-opt not incremental time(s) | (2) 3-opt incremental time(s) | ratio (1) / (2) time |
|---|---|---|---|
| a280 | 16.2 | 10.8 | 1.49 |
| d198 | 11.2 | 10.4 | 1.08 |
| gr229 | 90.8 | 59.1 | 1.53 |
| kroA200 | 75.1 | 68.2 | 1.10 |
| pr136 | 19.5 | 13.1 | 1.49 |
| rat195 | 22.5 | 17.8 | 1.26 |
| mean | 39.22 | 29.92 | 1.33 |

**Table 3** Comparison of solving times for mandatory Hamiltonian path constraint with 2-opt, 3-opt, 4-opt and 5-opt.

| | 2-opt | | 3-opt | | 4-opt | | 5-opt | |
|---|---|---|---|---|---|---|---|---|
| | time(s) | #bk | time(s) | #bk | time(s) | #bk | time(s) | #bk |
| a280 | 6.5 | 2,182 | 10.8 | 3,134 | 132.0 | 3,386 | 41,884.8 | 3,386 |
| ch150 | 2.1 | 644 | 1.7 | 392 | 3.0 | 392 | 246.9 | 392 |
| d198 | 12.6 | 6,062 | 10.4 | 3,694 | 32.3 | 3,694 | 4,839.9 | 3,694 |
| gr229 | 64.6 | 44,696 | 59.1 | 33,336 | 97.4 | 23,930 | 11,001.2 | 26,036 |
| kroA200 | 63.8 | 33,166 | 68.2 | 34,058 | 71.1 | 31,050 | 1,450.7 | 31,050 |
| pr136 | 16.2 | 13,886 | 13.1 | 8,598 | 54.5 | 6,496 | 8,183.5 | 6,496 |
| rat195 | 17.9 | 10,286 | 17.8 | 8,560 | 56.9 | 10,192 | 5,816.3 | 10,192 |
| mean | 26.2 | 15,846.0 | 25.9 | 13,110.3 | 63.9 | 11,305.7 | 10,489.0 | 11,606.6 |

the use of 4-opt and 5-opt degrades the solving times compared to 2-opt and 3-opt. Indeed, for 4-opt we observe a loss of a factor greater than 2. For 5-opt, we observe a loss of a factor greater than 400. Thus, the solving times and number of backtracks trade-off is not good when $k > 3$.

In Table 4, we show the gap between the MIP solver Concorde [1] and the state of the art CP solving method with the mandatory Hamiltonian path constraint with 3-opt. Note that the results for Concorde are obtained on our machine. For the small sized instances, we notice that our method is competitive with Concorde. Indeed, small sized instances such as att48 are solved in 0.14s with Concorde whereas we solved it in 0.03s. For medium sized instances such as rat195, we can see a slight degradation of the results: Concorde solved it in 8.73s whereas we solved it in 17.82s. However, the solving times are still comparable. Unfortunately, our method starts to slowing down for larger instances. For example, rd400 is solved in 20.6s with Concorde whereas it is solved in 12,351.85s with our method. Nevertheless, in [2], the solving time ratio with Concorde of kroC100 is about 1000, here it is only 3.1. Thus, we hope that same improvement factors will be obtained for larger instances in future works.

## 5 Conclusion

In this paper, we introduced a new constraint based on the $k$-opt algorithm, named mandatory Hamiltonian path constraint, into to the TSP model in CP. We also introduced an incremental version of this constraint. Experiments have shown that the use of this constraint leads to an

**Table 4** Comparison of the solving times and the number of backtracks for mandatory Hamiltonian path constraint with 3-opt and Concorde.

| | Concorde | | 3-opt | | ratio |
|---|---|---|---|---|---|
| | time(s) | #bk | time(s) | #bk | time |
| gr24 | 0.02 | 0 | 0.00 | 2 | 0.0 |
| att48 | 0.14 | 0 | 0.03 | 6 | 0.2 |
| eil51 | 0.07 | 0 | 0.05 | 32 | 0.8 |
| st70 | 0.12 | 0 | 0.14 | 70 | 1.1 |
| kroC100 | 0.35 | 0 | 1.08 | 334 | 3.1 |
| bier127 | 0.31 | 0 | 0.28 | 60 | 0.9 |
| gr137 | 1.32 | 0 | 1.55 | 518 | 1.2 |
| ch150 | 0.93 | 0 | 1.73 | 392 | 1.9 |
| si175 | 3.58 | 2 | 342.64 | 275,870 | 95.8 |
| rat195 | 8.73 | 6 | 17.82 | 8,560 | 2.0 |
| gr202 | 2.92 | 0 | 2.30 | 448 | 0.8 |
| lin318 | 2.59 | 0 | 10.12 | 2,018 | 3.9 |
| ali535 | 6.72 | 0 | 35,025.96 | 3,178,482 | 5215.3 |
| d493 | 47.17 | 4 | 31,162.09 | 1,877,298 | 660.6 |
| rd400 | 20.60 | 8 | 12,351.85 | 2,507,272 | 599.7 |

improvement of at least a factor of 3 in solving times. In addition, it shown that the use of 3-opt is well suited for our constraint. Moreover, we have been able to solve some instances that remains unsolved with the state of the art CP model. The $k$-opt algorithm is embedded in most of the solving methods of the TSP and therefore now in the CP. In future work, we will study an extension of this constraint not only considering the mandatory Hamiltonian paths but the mandatory cutsets in the graph.

## References


1 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

2 Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233, 2012. URL: `https://hal.archives-ouvertes.fr/hal-01344070`.

3 Jon Jouis Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on computing*, 4(4):387–411, 1992.

4 Václav Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Math. Program.*, 5(1):29–40, 1973. `doi:10.1007/BF01580109`.

5 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

6 Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. `doi:10.4153/CJM-1965-045-4`.

7 Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. The salesman and the tree: the importance of search in CP. *Constraints*, 21(2):145–162, 2016.

8 Martin Grötschel and Manfred Padberg. On the symmetric travelling salesman problem i: Inequalities. *Mathematical Programming*, 16:265–280, December 1979. `doi:10.1007/BF01582116`.

9 Robert Haralick and Gordon Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, January 1979.

10 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

11 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, 1971.

12 Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. doi:10.1016/S0377-2217(99)00284-2.

13 Nicolas Isoart and Jean-Charles Régin. Integration of structural constraints into tsp models. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, pages 284–299, Cham, 2019. Springer International Publishing.

14 Nicolas Isoart and Jean-Charles Régin. Adaptive CP-Based Lagrangian Relaxation for TSP Solving. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 300–316, Cham, 2020. Springer International Publishing.

15 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

16 Adam N. Letchford and Andrea Lodi. Polynomial-Time Separation of Simple Comb Inequalities. In William J. Cook and Andreas S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 93–108, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

17 S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21:498–516, 1973.

18 Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

19 Ilhan Or. Traveling salesman type combinatorial problems and their relation to the logistics of regional blood banking, 1977.

20 Gerhard Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

21 Robert E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.